FASTSUBS: An Efficient and Exact Procedure for Finding the Most Likely Lexical Substitutes Based on an N-gram Language Model

Deniz Yuret

***Abstract*—Lexical substitutes have found use in areas such as paraphrasing, text simplification, machine translation, word sense disambiguation, and part of speech induction. However the computational complexity of accurately identifying the most likely substitutes for a word has made large scale experiments difficult. In this paper I introduce a new search algorithm,** FASTSUBS**, that is *guaranteed* to find the K most likely lexical substitutes for a given word in a sentence based on an n-gram language model. The computation is sub-linear in both K and the vocabulary size V. An implementation of the algorithm and a dataset with the top 100 substitutes of each token in the WSJ section of the Penn Treebank are available at http://goo.gl/jzKH0.**

EDICS Category: SPE-LANG

I. Introduction

Lexical substitutes have proven useful in applications such as paraphrasing [1], text simplification [2], and machine translation [3]. Best published results in unsupervised word sense disambiguation [4], and part of speech induction [5] represent word context as a vector of substitute probabilities. Using a statistical language model to find the most likely substitutes of a word in a given context is a successful approach ([6], [7]). However the computational cost of an exhaustive algorithm, which computes the probability of every word before deciding the top K, makes large scale experiments difficult. On the other hand, heuristic methods run the risk of missing important substitutes.

This paper presents the FASTSUBS algorithm which can efficiently and correctly identify the most likely lexical substitutes for a given context based on an n-gram language model without going through most of the vocabulary. Even though the worst-case performance of FASTSUBS is still proportional to vocabulary size, experiments demonstrate that the average cost is sub-linear in both the number of substitutes K and the vocabulary size V. To my knowledge, this is the first sub-linear algorithm that exactly identifies the top K most likely lexical substitutes.

The efficiency of FASTSUBS makes large scale experiments based on lexical substitutes feasible. For example, it is possible to compute the top 100 substitutes for each one of the 1,173,766 tokens in the WSJ section of the Penn Treebank [8] in under 5 hours on a typical workstation. The same task would take about 6 days with the exhaustive algorithm. The Penn Treebank substitute data and an implementation of the algorithm are available from the author's website at http://goo.gl/jzKH0.

Section II derives substitute probabilities as defined by an n-gram language model with an arbitrary order and smoothing. Section III describes the FASTSUBS algorithm. Section IV proves the correctness of the algorithm and Section V presents experimental results on its time complexity. Section VI summarizes the contributions of this paper.

II. Substitute Probabilities

This section presents the derivation of lexical substitute probabilities based on an n-gram language model. Details of this derivation are important in finding an admissible algorithm that identifies the most likely substitutes efficiently, without trying out most of the vocabulary.

N-gram language models assign probabilities to arbitrary sequences of words (or other tokens like punctuation etc.) based on their occurrence statistics in large training corpora. They approximate the probability of a sequence of words by assuming each word is conditionally independent of the rest given the previous $(n-1)$ words. For example a trigram model would approximate the probability of a sequence $abcde$ as:

$$p(abcde) = p(a)p(b|a)p(c|ab)p(d|bc)p(e|cd) \quad (1)$$

where lowercase letters like a, b, c represent words and strings of letters like $abcde$ represent word sequences. The computation is typically performed using log probabilities, which turns the product into a summation:

$$\ell(abcde) = \ell(a) + \ell(b|a) + \ell(c|ab) + \ell(d|bc) + \ell(e|cd) \quad (2)$$

where $\ell(x) \equiv \log p(x)$. The individual conditional probability terms are typically expressed in back-off form:[1]

$$\ell(c|ab) = \begin{cases} \alpha(abc) & \text{if } f(abc) > 0 \\ \beta(ab) + \ell(c|b) & \text{otherwise} \end{cases} \quad (3)$$

where $\alpha(abc)$ is the discounted log probability estimate for $\ell(c|ab)$ (typically slightly less than the log frequency in the training corpus), $f(abc)$ is the number of times abc has been observed in the training corpus, $\beta(ab)$ is the back-off weight

D. Yuret is with the Department of Computer Engineering, Koç University, İstanbul, Turkey, e-mail: dyuret@ku.edu.tr.

[1]Even interpolated models can be represented in the back-off form and in fact that is the way SRILM stores them in ARPA (Doug Paul) format model files.

to keep the probabilities add up to 1. The formula can be generalized to arbitrary n-gram orders if we let b stand for zero or more words. The recursion bottoms out at unigrams (single words) where $\ell(c) = \alpha(c)$. If there are any out-of-vocabulary words we assume they are mapped to a special $\langle \text{UNK} \rangle$ token, so $\alpha(c)$ is never undefined.

It is best to use both left and right context when estimating the probabilities for potential lexical substitutes. For example, in "*He lived in San Francisco suburbs.*", the token *San* would be difficult to guess from the left context but it is almost certain looking at the right context. The log probability of a substitute word given both left and right contexts can be estimated as:

$$\begin{aligned} \ell(x|ab\_de) &\propto \ell(abxde) \\ &\propto \ell(x|ab) + \ell(d|bx) + \ell(e|xd) \end{aligned} \tag{4}$$

Here the "_" symbol represents the position the candidate substitute x is going to occupy. The first line follows from the definition of conditional probability and the second line comes from Equation 1 except the terms that do not include the candidate x have been dropped.

The expression for the unnormalized log probability of a lexical substitute according to Equation 4 and the decomposition of its terms according to Equation 3 can be combined to give us Equation 5. For arbitrary order n-gram models we would end up with a sum of n terms and each term would come from one of n alternatives.

$$\begin{aligned} \ell(x|ab\_de) \quad &\propto \\ &\begin{cases} \alpha(abx) & \text{if } f(abx) > 0 \\ \beta(ab) + \alpha(bx) & \text{if } f(bx) > 0 \\ \beta(ab) + \beta(b) + \alpha(x) & \text{otherwise} \end{cases} \\ + &\begin{cases} \alpha(bxd) & \text{if } f(bxd) > 0 \\ \beta(bx) + \alpha(xd) & \text{if } f(xd) > 0 \\ \beta(bx) + \beta(x) + \alpha(d) & \text{otherwise} \end{cases} \\ + &\begin{cases} \alpha(xde) & \text{if } f(xde) > 0 \\ \beta(xd) + \alpha(de) & \text{if } f(de) > 0 \\ \beta(xd) + \beta(d) + \alpha(e) & \text{otherwise} \end{cases} \end{aligned} \tag{5}$$

III. Algorithm

The task of FASTSUBS is to pick the top K substitutes (x) from a vocabulary of size V that maximize Equation 5 for a given context ab_de. Equation 5 forms a tree where leaf nodes are primitive terms such as $\beta(bx)$, $\alpha(xd)$, and parent nodes are compound terms, i.e. sums or conditional expressions. The basic strategy is to construct a priority queue of candidate substitutes for Equation 5 by composing substitute queues for each of its sub-expressions. The structure of these queues and how they can be composed is described next, followed by the construction of the individual queues for each of the subexpressions.

A. Upper bound queues

A sum such as $\beta(bx)+\alpha(xd)$ is not necessarily maximized by the x's that maximize either of its terms. What we can say for sure is that the sum for any x cannot exceed the upper bound $\beta(bx_1) + \alpha(x_2 d)$ where x_1 maximizes $\beta(bx)$ and x_2 maximizes $\alpha(xd)$. We can find the x that maximizes the sum by repeatedly evaluating candidates until we find one whose value is (i) larger than all the candidates that have been evaluated, and (ii) larger than the upper bound for the remaining candidates.

Based on this intuition, we define an abstract data type called an *upper bound queue* that maintains an *upper bound* on the actual values of its elements. Each successive *pop* from an upper bound queue is not guaranteed to retrieve the element with the largest value, but the remaining elements are guaranteed to have values smaller than or equal to a non-increasing upper bound. An upper bound queue supports three operations:

- $\text{SUP}(q)$: returns an upper bound on the value of the elements in the queue.
- $\text{TOP}(q)$: returns the top element in the queue. Note that this element is not guaranteed to have the highest value.
- $\text{POP}(q)$: extracts and returns the top element in the queue and updates the upper bound if possible.

Upper bound queues can be composed easily. Going back to our sum example let us assume that we have valid upper bound queues q_α for $\alpha(xd)$ and q_β for $\beta(bx)$. The queue q_σ for the sum $(\beta(bx) + \alpha(xd))$ has $\text{SUP}(q_\sigma) = \text{SUP}(q_\alpha) + \text{SUP}(q_\beta)$ because the upper bound for a sum clearly cannot exceed the total of the upper bounds for its constituent terms. $\text{TOP}(q_\sigma)$ can return any element from the queue without violating the contract. However in order to find the true maximum, we eventually need an element whose value exceeds the upper bound for the remaining elements. Thus we can bias our choice for $\text{TOP}(q_\sigma)$ to prefer elements that (i) have high values, and (ii) reduce the upper bound quickly. In practice non-deterministically picking $\text{TOP}(q_\sigma)$ to be one of $\text{TOP}(q_\alpha)$ or $\text{TOP}(q_\beta)$ works well. $\text{POP}(q_\sigma)$ can extract and return the same element from the corresponding child queue. If the upper bound of a child queue drops as a result, so does the upper bound of the compound queue q_σ.

B. Top level queue

The top level sum in Equation 5 is a sum of N conditional expressions for an order N language model. We can construct an upper bound queue for the sum using the upper bound queues for its constituent terms as described in the previous section. Let q represent the queue for the top level sum, $\delta \in C$ represent the constituent conditional expressions and q_δ represent their associated queues.

$$\begin{aligned} \text{SUP}(q) &= \sum_{\delta \in C} \text{SUP}(q_\delta) \\ \text{TOP}(q) &= \text{TOP}(q_\delta) \text{ for a random } \delta. \end{aligned} \tag{6}$$

For $\text{TOP}(q)$ we non-deterministically pick the top element from one of the children and $\text{POP}(q)$ extracts and returns that same element adjusting the upper bound if necessary.

As mentioned before $\text{TOP}(q)$ does not necessarily return the element with the maximum value. In order to find the top K elements FASTSUBS keeps popping elements from q and

computes their true values according to Equation 5 until at least K of them have values above the upper bound for the remaining elements in the queue. Table I gives the pseudo-code for FASTSUBS .

FASTSUBS (S, K)
1) Initialize upper bound queue q for context S.
2) Initialize set of candidate words $X = \{\}$.
3) WHILE $|\{x : x \in X, \ell(x|S) \geq \text{SUP}(q)\}| < K$ DO $X := X \cup \{\text{POP}(q)\}$
4) Return top K words in X based on $\ell(x|S)$.

TABLE I
PSEUDO-CODE FOR FASTSUBS . GIVEN A WORD CONTEXT S AND THE DESIRED NUMBER OF SUBSTITUTES K, FASTSUBS RETURNS THE SET OF TOP K WORDS THAT MAXIMIZE $\ell(x|S)$.

This procedure will return the correct result as long as $\text{POP}(q)$ cycles through all the words in the vocabulary and the upper bound for the remaining elements, $\text{SUP}(q)$, is accurate. The loop can in fact cycle through all the words in the vocabulary because at least one of the subexpressions, $\alpha(x)$, is well defined for every word. The accuracy of $\text{SUP}(q)$ depends on the accuracy of the upper bounds for constituent terms, which are described next.

C. Queues for conditional expressions

Conditional expressions indicated by "{" in Equation 5 pick their topmost child whose α argument has been observed in the training corpus. Let q_δ be the queue for such a conditional expression and $\sigma \in C_\delta$ be its children terms. Let $\sigma_{\max} = \arg\max_{\sigma \in C_\delta} \text{SUP}(q_\sigma)$ be the child whose queue has the maximum upper bound. The upper bound for q_δ cannot exceed the upper bound for $q_{\sigma_{\max}}$ because the value of the conditional expression for any given x is equal to the value of one of its children. Thus we define the queue operations for conditional expressions based on $q_{\sigma_{\max}}$:

$$\begin{aligned} \text{SUP}(q_\delta) &= \text{SUP}(q_{\sigma_{\max}}) \\ \text{TOP}(q_\delta) &= \text{TOP}(q_{\sigma_{\max}}) \end{aligned} \tag{7}$$

D. Queues for sums of primitive terms

As described in Section III-A, the upper bound of a queue for a sum like $\beta(bx) + \alpha(xd)$ is equal to the sum of the upper bounds of the constituent queues. It turns out that for sums of primitive terms, only the α term that has the candidate word x as an argument has a non-constant upper-bound. The language model defines β to be 0 for any word sequence that does not appear in the training set. Therefore the β terms that have the candidate word x as an argument always have the upper bound 0. Finally, the α and β terms without the candidate word x act as constants.

For notational consistency we define upper bounds for the constant terms as well. Let A and B represent sequences of zero or more words that do not include the candidate x. We have:

$$\begin{aligned} \text{SUP}(q_\alpha(A)) &= \alpha(A) \\ \text{SUP}(q_\beta(B)) &= \beta(B) \end{aligned} \tag{8}$$

For β terms with x in their argument, many words from the vocabulary would be unobserved in the argument sequence and share the maximum β value of 0. In the rare case that all vocabulary words have been observed in the argument sequence, they would each have negative β values and 0 would still be a valid upper bound. Thus FASTSUBS uses the constant 0 as an upper bound for β terms with x.

$$\text{SUP}(q_\beta(AxB)) = 0 \tag{9}$$

Only the α term with an x argument has an upper bound queue as described in the next section. FASTSUBS picks the top element for a sum of primitive terms only from its α constituent.[2] Let q_σ be the queue for a sum of primitive terms and let $\gamma \in C_\sigma$ indicate its constituents (α, β, constant or otherwise). We have:

$$\text{SUP}(q_\sigma) = \sum_{\gamma \in C_\sigma} \text{SUP}(q_\gamma) \tag{10}$$

$$\text{TOP}(q_\sigma) = \begin{cases} \text{TOP}(q_\alpha) \text{ if the } \alpha \text{ term has an } x \text{ argument.} \\ \text{UNDEF otherwise.} \end{cases}$$

E. Queues for primitive terms

FASTSUBS pre-computes actual priority queues (which satisfy the upper bound queue contract) for α terms that include x in their argument:

$$\begin{aligned} \text{SUP}(q_\alpha(AxB)) &= \max_x \alpha(AxB) \\ \text{TOP}(q_\alpha(AxB)) &= \arg\max_x \alpha(AxB) \end{aligned} \tag{11}$$

Here A and B stand for zero or more words and x is a candidate lexical substitute word. $\text{SUP}(q_\alpha)$ gives the real maximum, thus provides a tight upper bound. $\text{TOP}(q_\alpha)$ is guaranteed to return the element with the highest value.

The q_α queues are constructed once in the beginning of the program as sorted arrays and re-used in queries for different contexts. The construction can be performed in one pass through the language model and the memory requirement is of the same order as the size of the language model. Candidates that have not been observed in the argument context will be at the bottom of this queue because $\alpha(AxB) \equiv -\infty$ if $f(AxB) = 0$. To save memory such x are not placed in the queue. Thus after we run out of elements in q_α the queue returns:

$$\begin{aligned} \text{SUP}(q_\alpha(AxB)) &= -\infty \\ \text{TOP}(q_\alpha(AxB)) &= \text{UNDEF} \end{aligned} \tag{12}$$

IV. CORRECTNESS

As mentioned in Section III, the correctness of the algorithm depends on two factors: (i) the $\text{SUP}(q)$ function should return an upper bound on the remaining values in q, and (ii) the $\text{POP}(q)$ function should cycle through the whole vocabulary for the top level queue.

The correctness of the $\text{SUP}(q)$ function can be proved recursively. For primitive terms $\text{SUP}(q)$ is equal to the actual

[2] Remember that the top value in an upper bound queue is not guaranteed to have the largest value. Thus ignoring the β terms does not effect the correctness of the algorithm.

maximum (e.g. for q_α), or is an obvious upper bound (e.g. $\text{SUP}(q_\beta(AxB)) = 0$). For sums, $\text{SUP}(q)$ is equal to the sum of the upper bounds for the children and, for conditional expressions, $\text{SUP}(q)$ is equal to the maximum of the upper bounds for the children.

To prove that $\text{POP}(q)$ will cycle through the entire vocabulary it suffices to show that the queue for at least one child of q will cycle through the entire vocabulary. This is in fact the case because one of the children will always include the term $\alpha(x)$ whose queue contains the entire vocabulary.

V. Complexity

A exhaustive algorithm to find the most likely substitutes in a given context could try each word in the vocabulary as a potential substitute x and compute the value of the expression given in Equation 5. The computation of Equation 5 requires $O(N^2)$ operations for an order N language model, which we will assume to be a constant. If we have V words in our vocabulary the cost of the exhaustive algorithm to find a single most likely substitute would be $O(V)$.

In order to quantify the efficiency of FASTSUBS on a real world dataset, I used a corpus of 126 million words of WSJ data as the training set and the WSJ section of the Penn Treebank [8] as the test set. Several 4-gram language models were built from the training set using Kneser-Ney smoothing in SRILM [9] with vocabulary sizes ranging from 16K to 512K words. The average number of $\text{POP}(q)$ operations for the top level upper bound queue was measured for number of substitutes K ranging from 1 to 16K. Figure 1 shows the results.

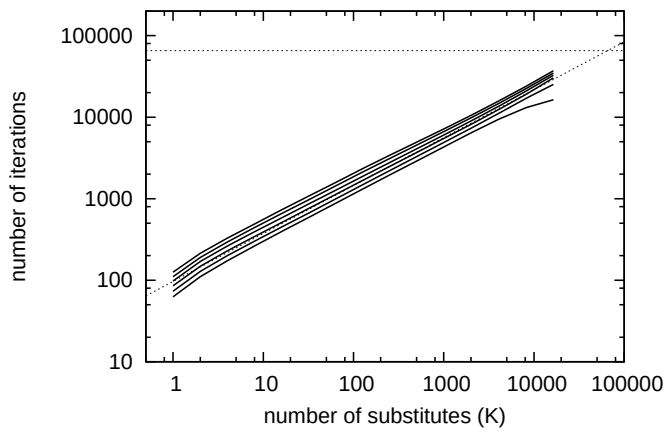

Fig. 1. Number of iterations as a function of the number of substitutes K and the vocabulary size V. The solid curves represent results with vocabulary sizes from 16K to 512K. The horizontal dotted line gives the cost of the exhaustive algorithm for $V = 64K$. The diagonal dotted line is a functional approximation in the form $K^\lambda V^{(1-\lambda)}$ for $V = 64K$ and $\lambda = 0.5878$.

The time cost of FASTSUBS depends on the number of iterations of the while loop in Table I which in turn depends on the quality of words returned by $\text{POP}(q)$ and the tightness of the upper bound given by $\text{SUP}(q)$. The worst case is no better than the exhaustive algorithm's $O(V)$. However Figure 1 shows that the average performance of FASTSUBS on real data is significantly better when $K \ll V$. The number of $\text{POP}(q)$ operations in the while loop to get the top K substitutes is sub-linear in K (the slope of the log-log curves are around 0.5878) and approaches the vocabulary size V as $K \to V$. The effect of vocabulary size is practically insignificant: increasing vocabulary size from 16K to 512K less than doubles the average number of steps for a given K.

As a practical example, it is possible to compute the top 100 substitutes for each one of the 1,173,766 tokens in Penn Treebank with a vocabulary size of 64K in under 5 hours on a typical 2012 workstation.[3] The same task would take about 6 days for the exhaustive algorithm.

VI. Contributions

Finding likely lexical substitutes has a range of applications in natural language processing. In this paper we introduced an exact and efficient algorithm, FASTSUBS , that is guaranteed to find the K most likely substitutes for a given word context from a V word vocabulary. Its average runtime is sub-linear in both V and K giving a significant improvement over an exhaustive $O(V)$ algorithm when $K \ll V$. An implementation of the algorithm and a dataset with the top 100 substitutes of each token in the WSJ section of the Penn Treebank are available at http://goo.gl/jzKH0.

Acknowledgments

I would like to thank the members of the Natural Language Group at USC/ISI for their hospitality and for convincing me that a FASTSUBS algorithm is possible.

References

[1] D. McCarthy and R. Navigli, "Semeval-2007 task 10: English lexical substitution task," in *Proceedings of the 4th International Workshop on Semantic Evaluations (SemEval-2007)*, 2007, pp. 48–53.

[2] L. Specia, S. Jauhar, and R. Mihalcea, "Semeval-2012 task 1: English lexical simplification," in *Proceedings of the International Workshop on Semantic Evaluation*, 2012, forthcoming.

[3] R. Mihalcea, R. Sinha, and D. McCarthy, "Semeval-2010 task 2: Cross-lingual lexical substitution," in *Proceedings of the 5th International Workshop on Semantic Evaluation*. Association for Computational Linguistics, 2010, pp. 9–14.

[4] D. Yuret and M. A. Yatbaz, "The noisy channel model for unsupervised word sense disambiguation," *Computational Linguistics*, vol. 36, no. 1, pp. 111–127, March 2010. [Online]. Available: http://www.aclweb.org/anthology-new/J/J10/J10-1004.pdf

[5] M. A. Yatbaz, E. Sert, and D. Yuret, "Learning syntactic categories using paradigmatic representations of word context," in *Proceedings of the 2012 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, July 2012.

[6] T. Hawker, "Usyd: Wsd and lexical substitution using the web1t corpus," in *SemEval-2007: 4th International Workshop on Semantic Evaluations*, 2007. [Online]. Available: /ref/hawker/98.pdf

[7] D. Yuret, "Ku: Word sense disambiguation by substitution," in *Proceedings of the 4th International Workshop on Semantic Evaluations*. Association for Computational Linguistics, 2007, pp. 207–213.

[8] M. P. Marcus, B. Santorini, M. A. Marcinkiewicz, and A. Taylor, *Treebank-3*. Philadelphia: Linguistic Data Consortium, 1999.

[9] A. Stolcke, "Srilm – an extensible language modeling toolkit," in *Seventh International Conference on Spoken Language Processing*, 2002.

[3] Running a single thread on an Intel Xeon E7-4850 2GHz processor.